

CSC 263 Lecture 5

June 22, 2004

For another reference, see chapter 11 of the textbook (CLRS).

1 Direct Addressing

Recall that a *dictionary* is an ADT that supports the following operations on a set of elements with well-ordered key-values: **INSERT**, **DELETE**, **SEARCH**. If we know the key-values are integers from 1 to K , for instance, then there is a simple and fast way to represent a dictionary: just allocate an array of size K and store an element with key i in the i th cell of the array.

This data structure is called *direct addressing* and supports all three of the important operations in worst-case time $\Theta(1)$. There is a major problem with direct addressing, though. If the key-values are not bounded by a reasonable number, the array will be huge! Remember that the amount of space that a program requires is another measure of its complexity. Space, like time, is often a limited resource in computing.

Example 1: A good application of direct addressing is the problem of reading a textfile and keeping track of the frequencies of each letter (one might need to do this for a compression algorithm such as Huffman coding). There are only 256 ASCII characters, so we could use an array of 256 cells, where the i th cell will hold the count of the number of occurrences of the i th ASCII character in our textfile.

Example 2: A bad application of direct addressing is the problem of reading a datafile (essentially a list of 32-bit integers) and keeping track of the frequencies of each number. The array would have to be of size 2^{32} , which is pretty big!

2 Hashing

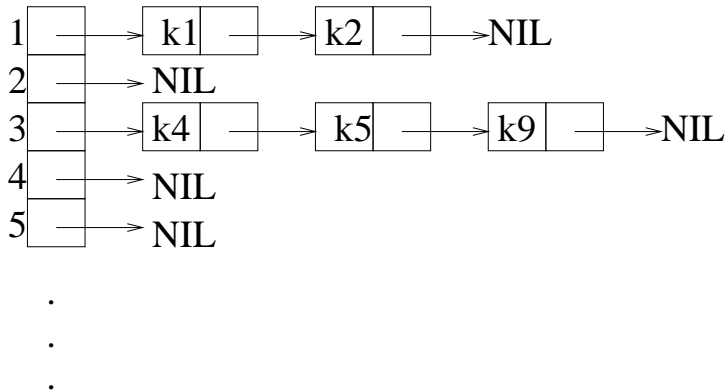
A good observation about example 2 or about any situation where the range of key-values is large, is that a lot of these might not occur very much, or maybe even not at all. If this is the case, then we are wasting space by allocating an array with a cell for every single key-value.

Instead, we can build a *hash table*: if the key-values of our elements come from a *universe* (or set) U , we can allocate a table (or an array) of size m (where $m < |U|$), and use a function $h : U \rightarrow \{0, \dots, m-1\}$ to decide where to store a given element (that is, an element with key-value x gets stored in position $h(x)$ of the hash table). The function h is called a *hash function*.

2.1 Closed Addressing

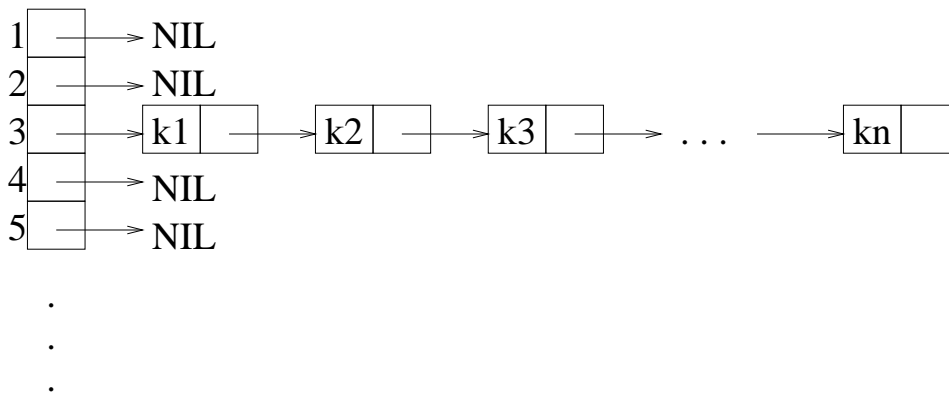
If $m < |U|$, then there must be $k_1, k_2 \in U$ such that $k_1 \neq k_2$ and yet $h(k_1) = h(k_2)$. This is called a *collision*; there are several ways to resolve it. One is to store a linked list at each entry in the hash

table, so that an element with key k_1 and an element with key k_2 can both be stored at position $h(k_1) = h(k_2)$ (see figure). This is called *chaining*.



Assuming we can compute h in constant time, then the INSERT operation will take time $\Theta(1)$, since, given an element a , we just compute $i = h(\text{key}(a))$ and insert a at the head of the linked list in position i of the hash table. DELETE also takes $\Theta(1)$ if the list is doubly-linked (given a pointer to the element that should be deleted).

The complexity of SEARCH(S, k) is a little more complicated. If $|U| > m(n - 1)$, then any given hash function will put at least n key-values in some entry of the hash table. So, the **worst case** is when every entry of the table has no elements except for one entry which has n elements and we have to search to the end of that list to find k (see figure). This takes time $\Theta(n)$ (not so good).



For the **average case**, the sample space is U (more precisely, the set of elements that have key-values from U). Whatever the probability distribution on U , we assume that our hash function h obeys a property called *simple uniform hashing*. This means that if A_i is the event (subset of U) $\{k \in U \mid h(k) = i\}$, then

$$\Pr(A_i) = \sum_{k \in A_i} \Pr(k) = 1/m.$$

In other words, each entry in the hash table is used just as much as any other. So the expected number of elements in any entry is n/m . We will call this the *load factor*, denoted by a . This assumption may or may not be accurate depending on U , h and the probability distribution on U .

To calculate the average-case running time, let T be a random variable which counts the number of elements checked when searching for key k . Let L_i be the length of the list at entry i in the hash

table. Then the average-case running time is:

$$E(T) = \sum_{k \in U} \Pr(k)T(k) \tag{1}$$

$$= \sum_{i=0}^{m-1} \sum_{k \in A_i} \Pr(k)T(k) \tag{2}$$

$$\leq \sum_{i=0}^{m-1} \Pr(A_i)L_i \tag{3}$$

$$= 1/m \sum_{i=0}^{m-1} L_i \tag{4}$$

$$= n/m \tag{5}$$

$$= a \tag{6}$$

So the average-case running time of `SEARCH` under simple uniform hashing with chaining is $O(a)$. Depending on the application, we can sometimes consider a to be constant since we can make m bigger when we know that n will be large. When this is the case, `SEARCH` takes time $O(1)$ on average.

2.2 Examples of Hash Functions

Recall the definition of simple uniform hashing: if A_i is the event (subset of U) $\{k \in U \mid h(k) = i\}$, then

$$\Pr(A_i) = \sum_{k \in A_i} \Pr(k) = 1/m.$$

Basically, the hash table gets evenly used for whatever distribution of keys we are dealing with. The problem is that we often don't know the distribution of keys before we see them. So how can we choose a good hash function?

For uniformly distributed keys in the range 1 through K (for large K), the following methods come close to simple, uniform hashing:

The division method: First choose a natural number m . Then, the hash function is just

$$h(k) = k \bmod m.$$

One advantage here is that computing $h(k)$ is very fast (just one division operation). But m has to be chosen with some care. If $m = 2^p$, then $h(k)$ is just the p lowest bits of k (see example 1). Instead, m is usually chosen to be a prime not close to any power of 2.

- **Example:** Most compilers or interpreters of computer programs construct a symbol table to keep track of the identifiers used in the input program. A hash table is a good data structure for a symbol table: identifiers need to be inserted and searched for quickly. We would like to use the division method for hashing, but first we need to turn the identifiers (strings of text) into positive integers. We can do this by considering each string to be a number in base 128 (if there are 128 text characters). Each character x can be represented by a number from 1 through 128 denoted $num(x)$. Then, a string of characters $x_n x_{n-1} \dots x_1$ can be represented uniquely by the number $\sum_{i=1}^n num(x_i)(128)^{i-1}$. For our choice of m here, we definitely want to avoid powers of 2, especially powers of 128. If m is 128^3 , for instance, then any two

identifiers that share the same last three letters will hash to the same entry in the table. If the program is computing a lot of maximum values, for instance, then many of the variable names may end in “max” and they will all collide in the hash table, causing longer search times if we use chaining.

The multiplication method: Another way to hash natural numbers is just to scale them to something between 0 and $m - 1$. Here we choose m (often a power of 2 in this case) and a real number A (often the fractional part of a common irrational number, such as the golden ratio: $(\sqrt{5} - 1)/2$). We then compute

$$h(k) = \lfloor m \times \text{fract}(kA) \rfloor,$$

where $\text{fract}(x)$ is the fractional part of a real number x . What’s the problem if A is “very” rational, like $\frac{1}{2}$?

2.3 Open Addressing

In *closed addressing*, we handled collisions by enlarging the storage capacity at the relevant entry in the hash table (in particular, we did this using a linked-list). In *open addressing*, each entry in the hash table stores only one element (so, in particular, we only use it when $n < m$). If we try to insert a new element and we get collision, then we have to look for a new location to store the new element. But we have to put it somewhere where we can find it if we’re searching for it. To insert it, we check a well-defined sequence of other locations in the hash table until we find one that’s not full. This sequence is called a *probe sequence*.

Linear Probing: The easiest open addressing strategy is linear-probing. For a hash table of size m , key k and hash function $h(k)$, the probe sequence is calculated as:

$$s_i = (h(k) + i) \bmod m \quad \text{for } i = 0, 1, 2, \dots$$

Note that s_0 (the home location for the item) is $h(k)$ since $h(k)$ should map to a value between 0 and $m - 1$.

What is the problem with linear probing? Clustering: As soon as we hash to something within a group of filled locations, we have to probe the whole group until we reach an empty slot and in doing so we increase the size of the cluster. Two keys that didn’t necessarily share the same “home” location end up with almost identical probe sequences.

Non-Linear Probing: Non-linear probing includes schemes where the probe sequence does not involve steps of fixed size. Consider quadratic probing where the probe sequence is calculated as:

$$s_i = (h(k) + i^2) \bmod m \quad \text{for } i = 0, 1, 2, \dots$$

There is still a problem, though: probe sequences will still be identical for elements that hash to the same home location.

Double Hashing: In double hashing we use a different hash function $h_2(k)$ to calculate the step size. The probe sequence is:

$$A_i = (h(k) + j * h_2(k)) \bmod m \quad \text{for } j = 0, 1, 2, \dots$$

Note that $h_2(k)$ shouldn’t be 0 for any k . Also, we want to choose h_2 so that, if $h(k_1) = h(k_2)$ for two keys k_1, k_2 , it won’t be the case that $h_2(k_1) = h_2(k_2)$. That is, the two hash functions don’t cause collisions on the same pairs of keys.

Analysis of Open Addressing: We'll look at the complexity of INSERT since, in open addressing, searching for a key k that is in the table takes exactly as long as it took to insert k in the first place. The time to search for an element k that does not appear in the table is the time it would take to insert that element in the table. You should check why these two statements are true.

It's not hard to come up with worst-case situations where the above types of open addressing require $\Theta(n)$ time for INSERT. On average, however, it can be very difficult to analyze a particular type of probing. Therefore, we will consider the following situation: there is a hash table with m locations that contains n elements and we want to insert a new key k . We will consider a random probe sequence for k —that is, its probe sequence is equally likely to be any permutation of $(0, 1, \dots, m-1)$. This is a realistic situation since, ideally, each key's probe sequence is as unrelated as possible to the probe sequence of any other key.

Let T denote the number of probes performed in the INSERT. Let A_i denote the event that every location up until the i -th probe is occupied. Then, $T \geq i$ iff A_1, A_2, \dots, A_{i-1} all occur, so

$$\begin{aligned} \Pr(T \geq i) &= \Pr(A_1 \cap A_2 \cap \dots \cap A_{i-1}) \\ &= \Pr(A_1) \Pr(A_2|A_1) \Pr(A_3|A_1 \cap A_2) \dots \Pr(A_{i-1}|A_1 \cap \dots \cap A_{i-2}) \end{aligned}$$

For $j \geq 1$,

$$\Pr(A_j|A_1 \cap \dots \cap A_{j-1}) = (n - j + 1)/(m - j + 1),$$

because there are $n - j + 1$ elements that we haven't seen among the remaining $m - j + 1$ slots that we haven't seen. Hence,

$$\Pr(T \geq i) = n/m \cdot (n - 1)/(m - 1) \dots (n - i + 2)/(m - i + 2) \leq (n/m)^{i-1} = a^{i-1}. \quad (7)$$

Now we can calculate the expected value of T , or the average-case complexity of insert:

$$E(T) = \sum_{i=0}^{m-1} i \Pr(T = i) \quad (8)$$

$$\leq \sum_{i=1}^{\infty} i \Pr(T = i) \quad (9)$$

$$= \sum_{i=1}^{\infty} i (\Pr(T \geq i) - \Pr(T \geq i + 1)) \quad (10)$$

$$= \sum_{i=1}^{\infty} \Pr(T \geq i) \quad (11)$$

$$= \sum_{i=1}^{\infty} a^{i-1} \quad \text{by (7)} \quad (12)$$

$$= \sum_{i=0}^{\infty} a^i \quad (13)$$

$$= \frac{1}{1 - a}. \quad (14)$$

Remember that $a < 1$ since $n < m$. The bigger the load factor, however, the longer it takes to insert something. This is what we expect, intuitively.